Due Thursday, 2000 Jan 27, at the beginning of class.

1 Revisions

2000Jan26: Revised unary.h to correct `operator>>()`. The old code did not correct detect end of input.

2000Jan24: Revised unary.h to remove the `size_type` code in `addOneToVector`. `size_type` is probably implemented as an integer and the homework forbids that.

There is probably no effect on your code because `addOneToVector` is called by `operator<<`, both of which were given to you and did not need modification.

2 Reading

Read chapter 9 of the textbook.

3 UNIX File Commands and File Security

Unless you take explicit action, anyone with a computer science account can look at all of your files in your CS account. This also means anyone can copy all of your files, submitting them as homework solutions. Do not fall victim to this form of identity stealing. Secure your own files and then ask a friend to check if you did it correctly.

4 Programming Tips

4.1 Compilation Tip

Last week, we mentioned three different command-line options for g++: `-Wall`, `-pedantic`, and `-o`. It is a pain to type these every compilation. Instead, one can store compilation commands in a "Makefile" and use the `make` command to compile your program, creating the executable.

For example, suppose you use the command

```
g++ -Wall -pedantic foo.cc -o foo
```

to compile a C++ file named "foo.cc" and create an executable called "foo." Instead, you can create a file called "Makefile" containing these two lines of code:

```
foo:
tab g++ -Wall -pedantic foo.cc -o foo
```

tab denotes a tab character; using eight spaces will not work correctly. To create the executable called "foo," type

```
make foo
```

The "make" programming language can automate many tasks. If you want to learn more, read the online GNU make documentation, particularly the first chapter.

4.2 Xemacs Tips

For this class, feel free to use any text editor you desire. I use Xemacs because, I have observed that most computer professionals use Xemacs or its less-friendly version Emacs. If you do not want to use Xemacs, just skip this section.

The Emacs tutorial discusses moving the cursor, dealing with files, etc. To run it, type 'Control-h' and then 't'.

To compile your program within Xemacs, click the Compile icon. The first time you do this, it will ask for the shell command to compile. For example, this could be "make foo" or "g++ -Wall -pedantic foo.cc -o foo". For every unsaved file, it will also ask if you want it saved.

The compiler's output will appear in another buffer. To visit each error, type Control-x ` (backtick). The cursor will move to the offending line in the C++ file, and the error will appear at the top of the compilation buffer. To visit a particular error, move the cursor to message in the compilation buffer and hit enter.

When running Xemacs via telnet, much of the nice user interface is not available, but all of Emacs's functionality is available. To compile, type "M-x compile". To compile using the same command, type "M-x recompile".

5 Warm-up Exercises

There is no need to submit solutions to these exercises, but be sure you can solve them.

1. Convert the pseudocode member() function, a.k.a. check-list() or find(), presented in class to C++ code.

2. Rewrite the member() function as a "one-liner," i.e., using only one line for the function's body.

   Hint: The C++ boolean operators `||` and `&&` use *short-circuit evaluation*, i.e., they evaluate only as many arguments as necessary (textbook, p. 107). For example, `false && foo()` need not evaluate `foo()` because the result is false.

6 Problem Statement

Warning! Understanding the recursive definition of unary numbers is fundamental to this homework. Please do not wait until Wednesday evening to determine whether you understand the definition. **End Warning.**

According to a former Stanford graduate student now a professor at Columbia University, the database program running on the main Stanford mainframe computer has no built-in arithmetic operations. Even though numbers such as tuition and housing charges were computed by this program, the program had no integer or floating point number type.

Our task is to implement arithmetic on natural numbers, i.e., nonnegative integers, to use with this program. Let's use unary number notation. Each number n is represented by the repeating a character n times. For example, the decimal number 7 would be represented by "0000000" and 3 would be represented by "000". Zero is represented by no digits: "".

We can recursively define any natural number as

- either zero

- or one more than another natural number.

For example, "000" is one more than one more than one more than zero. More succinctly, "000" is add1(add1(add1(zero))).

Note this recursive definition for numbers is very similar to the recursive definition of lists.

Since we assume that the database program supports lists of booleans, we can implement unary arithmetic using the linked list class discussed in class. Our database program must compute tuition charges so it needs $+$, $-$, $*$, integer division /, and exponentiation ˆ (to compute interest on overdue balances). Our database program will never use subtraction to produce negative numbers, but it needs to be able to compare numbers, i.e., $>$, $\geq$, $<$, $\leq$, $==$, and $!=$. You need not define arithmetic assignment operators such as $+=$ and $-=$, but your code may not use them either.

I have provided some source code (unary.h). For example, it defines

- the word `natural` as the type for our numbers,
- the number zero, and
- input and output functions. For example, `cin >> n` will read a base-10 number from the standard input, storing it into the (unary) natural number n. `cout << n` will print the base-10 representation of the (unary) natural number n.

  These input and output functions assume the following functions are defined:

  - The constructor `add1()` creates a larger number.
  - Given a nonzero number, the selector `sub1()` extracts its subnumber.
  - The predicate `iszero()` yields true if the number is zero.
  - The + and * operators are defined for natural numbers.

  These functions should be declared or defined before the input and output functions are defined.

For this assignment, write the omitted functions and add the other necessary functions to `unary.h`.

I have also provided a simple main() function (test-unary.cc) for preliminary testing of your code. Of course, you will want to write more extensive testing code.

6.1 Operator Overloading

For our natural numbers, we would like to be able to write expressions using the usual infix operators. For example, if `m` and `n` have type `natural`, we would like to be able to write expressions such as `n + m` and `n == m`. But since `m` and `n` are instances of a user-defined type rather than a built-in C++ type (such as `int` or `double`), the compiler has no idea how to evaluate such expressions. Instead, it converts these expressions to function calls, converting `m + n` into

```
operator+ (m, n)
```

and `m == n` into

```
operator== (m, n) .
```

We can thus ensure that `m + n` and `m == n` have the meanings we desire by writing functions with the following prototypes:

```
natural operator+ (const natural & m, const natural & n);
bool operator== (const natural & m, const natural & n);
```

The first function should return a `natural` number representing the sum of its parameters; the second function should return true if its parameters are equal and false otherwise. By providing such functions, we have "overloaded" the + and == operators. We can overload other operators in a similar fashion. See also Section 2.5 of the textbook.

6.2 Dealing with Multiple C++ Files

Fitting all the source code for complicated programs into one file would be unrealistic so C++ supports storing code in multiple files. Historically, files ending with ".cc" stored definitions, e.g., function definitions, while files ending with ".h" stored the corresponding declarations. Unfortunately, recent C++ changes have obfuscated these distinctions so sometimes we put function definitions in header files as we have done with `./unary.h`.

For this assignment, I provide these suggestions:

- Make sure that the files `seq.cc`, `seq.h`, `unary.h`, and `main.cc` are in the same directory.
- Do not change the given #include header directives, but feel free to add to them as necessary.
- Compile using the command `g++ -Wall -pedantic main.cc`. This will produce an executable named `a.out`. You can use the `-o` *`output-file-name`* g++ option if you desire.

Note the code, as distributed, should compile albeit with numerous warnings. Also, it will not run until code for the necessary functions is written. Later in the course, we will discuss the rules for dealing with multiple files.

7 Submission Rules

Submit a solution for the homework problem, not the warm-up exercises. Please email only one file: the `unary.h` file. You need not send `seq.h`, `seq.cc`, or `test-unary.cc`. When stress-testing your code, I will provide these files.

See the submission details for information how to email the programs.